Perl Programming Fundamentals for the Computational Biologist

Class 1

Marine Biological Laboratory
"Advances in Genome Technology and Bioinformatics"
Fall 2004

Andrew Tolonen
Chisholm Lab, MIT

Acknowledgements:

Penny Chisholm, MIT
Simon Kasif, Boston Univ.
George Church, Harvard Med. School
Art Newhall, WHOI
Lance Davidow, Mass. Gen. Hospital

Goals of the course:

- Introduce Perl (syntax, structure)

- Share some useful scripts to use as templates for your own projects

- Empower you as hackers
  - your computer is a lab tool to test hypotheses (just like the PCR machine)
  - the open source developers community is an incredibly powerful resource – even more powerful than the Microsoft tech support line!
    - Perl news groups:
      - comp.lang.perl.moderated
      - alt.perl
    - CPAN (Comprehensive Perl Archive Network): www.cpan.org
    - My email: tolonen@mit.edu

Outline of the Course

Class 1: Intro to Perl
- Data types: scalars, arrays, hashes
- Control structures: if...else, while, for, foreach
- Filehandles (I/O)
- Intro to Pattern Matching

Class 2: Regular expressions
- Subroutines
- Modules
- Examples of common problems in bioinformatics

What is Perl?

“Perl is the duct tape of the internet”

- Randal Schwartz (author of Learning Perl)

“TMTOWTDI.”

- Wall, Christiansen and Orwant, Programming Perl

“Perl is the Cliff Notes of Unix”.

- Larry Wall

“Perl is optimized for problems which are about 90% working with text and about 10% everything else.”

- (Schwartx and Phoenix, Learning Perl)

“The very definition of hell is having to maintain someone else's Perl code.”

- Anonymous

$happiness =~ s/Perl/Java/g;

- adapted from Jamie Zawinski

Does my computer have Perl?

Determine what version of Perl is installed

```
% perl -v
This is perl, v5.8.4 built for sparc-linux-thread-
multi
```

Locating the Perl executable

```
% whereis perl
perl: /usr/bin/perl
```

################################

If not, how do I get Perl?

Download it from www.cpan.org, It's free!

How do I run a Perl script?

Running a Perl script from a file:

```
% emacs myScript.plx

#!/usr/bin/perl

print “Hello world!\n”;  # that's it!

% chmod +x ./myScript.plx
make the perl script executable

% ./myScript.plx
```

Running a Perl script from the command line:

```
% perl -e 'print “Hello world!\n”';
```

Perl Variables

- A variable is just a box to hold your data
- Perl just has three data structures (types of boxes to hold your data)
    - scalars (variables denoted with $, i.e. $name)
    - arrays (variables denoted with @, i.e. @names)
    - hashes (variables denoted with %, i.e. %names)

################################

$Scalars

$$\begin{tikzcd} \$\text{name} \arrow[r] & \boxed{\text{Andy}} \end{tikzcd}$$

Examples of scalar variables are strings and numbers

```
$organism = “Escherichia coli K12”;
$genomeSize = 4639675;
```

use double-quotes to allow control chars (\n, \t) and variables within the quotes. use single-quotes to get string literals.

Scalar operators:

Comparing scalars

- compare numbers (==, <,<=) but don't use the single equals (=), that's for assignment!
- comparing strings (eq, ne, etc.)

Concatenating two scalars: you can concatenate two scalars using a dot (.).

```
$string1 = "Felicit";
$string2 = "ations";
$string3 = $string1.$string;
```

chomp() removes newlines from the end of a scalar variable

```
chomp($name);
```

substr() takes a scalar, a start position, and a length and extracts a substring. substr($string, offset, length).

```
$sequence = tgcattgtttactaca;
$subseq = substr($sequence, 0, 4); # grab the first
                     # four bases in the sequence
$subseq2 = substr($sequence, 4, 8); # grab bases 5
                 # through 12
```

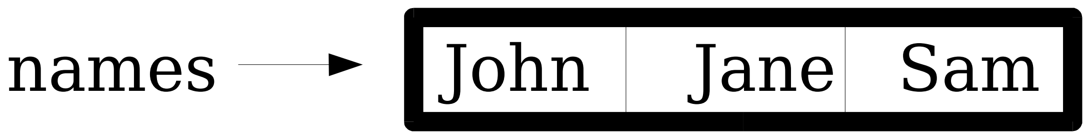
An example script using scalar variables:

```
#!/usr/bin/perl

$radius = 5;
$pi = 3.1415;
$circumference = 2 * $pi * $radius;

print “the circumference of the circle is
$circumference\n”;
```

################################

@Arrays

An array is a list of scalar data indexed with numbers. The first element of an array has an index = 0.



creating an array:

```
@names = (“John”, “Jane”, “Sam”);
```

accessing an element in the array

```
$secondName = $names[1];
```

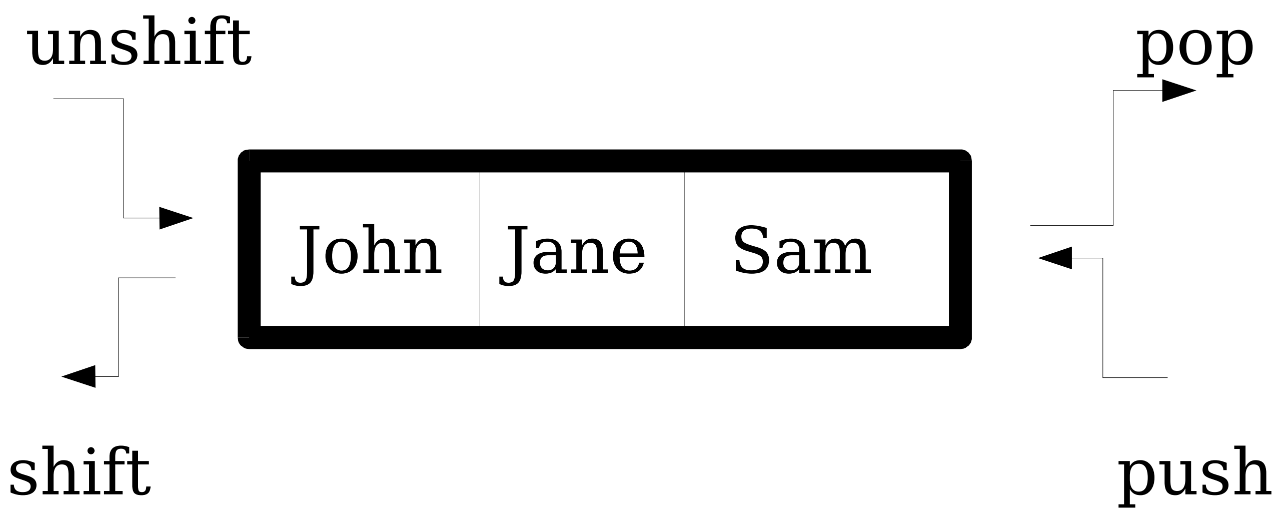
assigning a new element to an array

```
$names[3] = “Harry”;
```

printing an array

```
print “@names”;
```

Array operators



use unshift() to add an element to the front of an array

```
unshift(@names, “Andy”);
```

use shift() to remove an element from the front of an array

```
$name = shift(@names);
```

use push() to add an element to the end of an array

```
push(@names, “Greg”);
```

use pop() to remove an element from the end of an array

```
$name = pop(@array);
```

use reverse() to reverse the positions of the elements in an array

```
@reverseNames = reverse(@names);
```

use split() to break up a scalar into an array based on a pattern

```
$sentence = “Hi my name is Andy”;
@words = split(“ “, $sentence);
```

use <u>sort()</u> to sort the elements of an array in ascending ascii order (like a dictionary)

```
@sortedNames = sort(@names);
```

Note: you need different syntax to sort an array of numbers (i.e so 12 doesn't come before 6!)

```
@sortedNums= sort({$a<=>$b} @numbers);
```

##

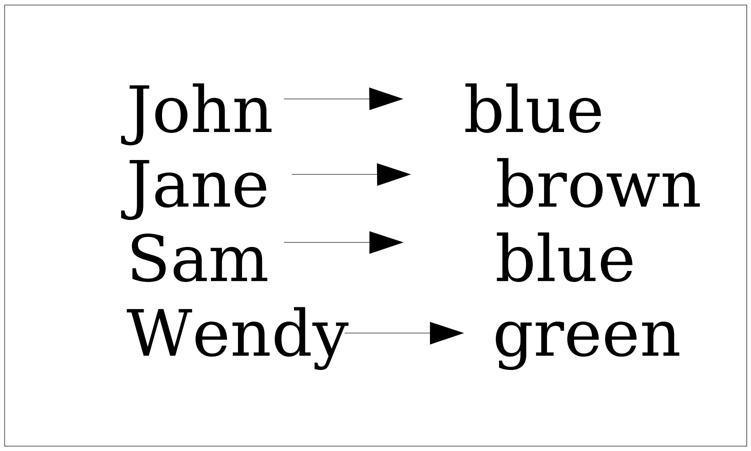
%Hashes

- Like arrays, a hash is a collection of scalars. But hashes use strings as indices instead of numbers. The indices are called keys. Keys point to values

  key => value

- Hashes are like barrels of data where each piece of data has a tag attached.
- The elements are in no particular order.
- The keys must be unique. Values can be duplicated.
-

%eyeColor



Create a hash

```
%genes = (“ntcA” => “transcription factor”, “pstS”
=> “phosphate stress sensor”, “rnpB” => “RNAse P”);
```

Access an element in a hash

```
$geneName = “ntcA”;
$function = $genes{$geneName};
```

Add an element to a hash

```
$geneName = “rpoC”;
$genes{$geneName} = “RNA polymerase subunit”;
```

Printing a hash

```
foreach $key (keys(%genes))
{
 print “$key  $genes{$key}\n”;
}
```

Hash operators

<u>keys()</u>: returns an array of all the keys of the hash. se e above.

<u>delete</u>: remove a key-value pair from a hash

###

Control Structures

Control structures are how to impose logic on your programs

<u>“If” loops</u>: execute once if an expression is true

```
if ($beerLabel eq “Harpoon”)
{
 print “Sure.  I'd love to come over for a
beer.\n”;
}
else
{
 print “Oh. Sorry, I really need to do laundry.\n”;
}
```

<u>“While” loops</u>: repeatedly execute so long as an expression is true

```
$bottles = 99;

while ($bottles >= 0)
```

```
{
 print “there are $bottles bottles of beer on the
wall\n”;
 $bottles = $bottles -1;
}
```

<u>“For” loops</u>: execute repeatedly so long as a test is true. for loops have the following syntax:

```
for ($bottles = 99; $bottles >= 0; $bottles--)
{
  print “there are $bottles bottles of beer on the
wall\n”;
}
```

<u>“Foreach” loops</u>: execute as many times as there are elements in @someList

```
foreach $element (@array)
{
  print “$element\n”;
}
```

##

Filehandles: Basic I/O (Input/Output)

- A filehandle is the name for an I/O connection between your Perl script and the outside world

- By convention, name your filehandles in uppercase
- There are 6 special filehandles in Perl. The most commonly used one STDIN, the filehandle that

connects your Perl script to the keyboard.

Prompt the user for input to your perl script:

```
#!/usr/bin/perl

print “What is your name?: “;
$name = <STDIN>;
chomp($name);
print “Your name is $name\n”;
```

Filehandle operators:

open() will open a filehandle.

open a file for input to your script
```
open(IN, “<./input.txt”);
```

open a file for output from your script
```
open(OUT, “>./output.txt”);
```

open a file for output, appending to the existing file
```
open(OUT, “>>./output.txt”);
```

close() will close a filehandle. If you forget to close a filehandle, Perl will do it automatically at the end of your script.

```
close(IN);
```

die will abort program with a fatal error if the filehandle cannot be opened.

```
open(IN, “<./input.txt”) or die “can't open input
file\n”;
```

Here is how you could open a file and read it into a variable called $file in your script.

```
#!/usr/bin/perl

open(IN, “<./Ecoli.faa”) or die “can't open input
file\n”;

while ($line = <IN>);
{
 $file = “$file”.”$line”;
}
########################################
```

Intro To Pattern Matching

One of Perl's greatest strengths is its ability to search for patterns in large files. It does this using regular expressions. Today we will introduce pattern matching.

The standard way to specify a pattern is to enclose it in forward slashes //. Variables are compared to a pattern using the binding operator =~.

```
if ($string =~ /pattern/)
{
  # do something;
}
```

In addition to just matching a pattern, you can make a substitution in a variable using a pattern.

```
$string =~ s/John/Jane/;
```

Here is an example:

```
#!/usr/bin/perl

$protein=”MPSLATLLIYLMAGTALGLLTLRTGIPAAPLAGA
LLGAGLVSMTGRLDVAEWPSGTRTAIEIAIGTVIGTGLTKSSLG
ELQNLWKPALLITLTLVLTGIVIGLWSSRLLGIDPVVSLLGAAP
GGISGMSLVGAEFGVGAAVATLHAVRLITVLLVLPLLVKLLAP”;

$protein =~ s/\n//g;  # remove all the newlines
$protein =~ s/\s//g;  # remove white space

$motif = “GGISG”;   # define the motif you are
                    # looking for

if ($protein =~ /$motif/)
{
print “Yeah. This protein contains the motif
$motif\n”;
}
```